Diamond: A Storage Architecture for Early Discard in Interactive Search

L. Huston, R. Sukthankar, R. Wickremesinghe,
M. Satyanarayanan, G. Ganger, E. Riedel, A. Ailamaki

IRP-TR-03-09
October 2003

Research at Intel

Diamond: A Storage Architecture for Early Discard in Interactive Search

L. Huston,[†] R. Sukthankar,[†•] R. Wickremesinghe,[†‡] M. Satyanarayanan,[†•]
G. Ganger,[•] E. Riedel,[⋆] A. Ailamaki[•]
[†]Intel Research Pittsburgh, [•]Carnegie Mellon University, [‡]Duke University, [⋆]Seagate Research

Abstract

This paper introduces the concept of *early discard* for interactive search of unindexed data. Processing data directly inside active storage devices using downloaded *searchlet* code enables Diamond to perform efficient, application-specific filtering of large data collections. Early discard helps users who are looking for "needles in a haystack" by eliminating the bulk of the irrelevant items as early as possible. A searchlet consists of a set of application-generated filters that Diamond uses to determine whether an object may be of interest to the user. The system optimizes the evaluation order of the filters based on run-time measurements of each filter's selectivity and computational cost. Diamond can also dynamically partition computation between the storage devices and the host computer to adjust for changes in hardware and network conditions. We provide an analysis of the behavior of our system and present performance numbers from a Linux-based prototype showing that Diamond can dynamically adapt to a query and run-time system state. An informal user study of an image retrieval application supports our belief that early discard significantly improves the quality of interactive searches.

1 Introduction

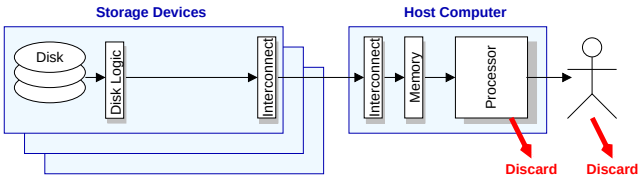
How does one find a few desired items in many terabytes or petabytes of complex and loosely-structured data such as digital photographs, video streams, CAT scans, AutoCAD drawings, or USGS maps? If the data has already been indexed for the query being posed, the problem is easy. Unfortunately, a suitable index is not always available. In that case, a user has no choice but to perform an exhaustive search over the entire volume of data. Although attributes such as the author, date, or other context of data items can restrict the search space, the user is still left with an enormous number of items to examine. Today, scanning such a large volume of data is so slow that it is only performed in the context of well-planned data mining. This is typically a batch job that runs overnight and is only rarely attempted interactively [16].

Our goal is to speed up scanning of large volumes of data so that interactive search of non-indexed data becomes practical. We believe that a key requirement for such speedup is the ability to discard irrelevant data items very close to their storage locations, rather than close to the user after they have passed through most of the system. We refer to this ability as *early discard*. We have developed a storage architecture and programming model called *Diamond* that embodies early discard. Diamond has been designed to run on an active disk [1, 19, 22] platform, but is not dependent on the availability of active storage devices. It can be realized using diverse storage back ends ranging from emulated active disks on a general-purpose cluster to storage nodes on a wide-area network.

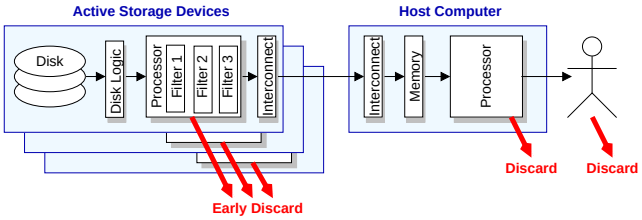
The remainder of the paper is organized as follows. Section 2 introduces the concept of early discard. Section 3 motivates why early discard is valuable for exhaustive search applications. Section 4 details the Diamond architecture. Section 5 describes a proof-of-concept image retrieval application using the Diamond system and presents an informal user study validating early discard in this context. Section 6 discusses implementation details, focusing on the dynamic partitioning of computation and the automatic re-ordering of filters. Section 7 presents experimental results for our Diamond implementation. Section 8 summarizes the related work in this area. Section 9 concludes and identifies future directions for this research.

2 Background and Motivation

The standard approach to efficient interactive search is to create an offline index of the data. Indexing assumes that the mapping between the user's query and the relevant data can be pre-computed, enabling the system to efficiently organize the data on the storage device so that only a small fraction of the data is accessed during a particular search. Unfortunately, indexing complex data remains a challenging problem for several reasons. First, manual indexing is often infeasible for large datasets and automated methods for extracting the semantic content from such data are still rather primitive (this is referred to in the literature as the *semantic gap* [20]). Second, the richness of the data often requires a high-dimensional representation that is not amenable to efficient indexing (a consequence of the *curse of dimensionality* [31, 6, 10]). Third, realistic user queries can be very sophisticated, requiring a great deal of domain knowledge that is often not available to the system for optimization. Finally, expressing the user's needs in a usable form can be extremely difficult (*e.g.*, "I need a photo of an energetic puppy playing with a happy toddler"). All of these problems limit



(a) Traditional storage architecture



(b) Diamond architecture

Figure 1. Unlike traditional architectures for exhaustive search, where all of the data must be shipped from to the host computer, the Diamond architecture employs *early discard* to efficiently reject the bulk of the irrelevant data at the active storage device.

the usability of *interactive data analysis* [16] today.

Figure 1(a) shows the traditional architecture for exhaustive search. Each data item passes through a pipeline from the disk surface, through the disk logic, over an interconnect to the host computer's memory. The search application can reject some of the data before presenting the rest to the user. Two problems with this design are: (1) the system is unable to take full advantage of the parallelism at the storage devices; (2) although the user is only interested in a small fraction of the data, all of it must be shipped from the storage devices to the host machine, and the bulk of the data is then discarded in the final stages of the pipeline. This is undesirable because the irrelevant data will often overload the interconnect or host processor.

Early discard is the idea of rejecting irrelevant data as early in the pipeline as possible. For instance, by exploiting active storage devices, one could eliminate a large fraction of the data before it was sent over the interconnect, as shown in Figure 1(b). The problem is that the storage device cannot determine the set of irrelevant objects *a priori* — the knowledge needed to recognize the useful data is only available to the search application (and to the user). However, if one could imbue some of the earlier stages of the pipeline with a portion of the intelligence of the application (or the user), exhaustive search would become much more efficient. This is supported by our experience with an interactive image search application, as described in Section 5.2.

2.1 Interactive Search

For most real-world applications, the sophistication of the user's query outpaces the development of algorithms that can understand the complex and domain-dependent data. For instance, in a homeland security context, state-of-the-art algorithms can reliably discard images that do not contain human faces, but face recognition software has not advanced to the point where it can recognize photos of particular individuals with sufficient accuracy. Thus, we believe that the majority of exhaustive search tasks will be interactive in nature, and it is important for systems to consider the human as an important stage in the pipeline. In particular, a good system design should model the user's limited rate of data processing, and should tune the aggressiveness of early discard to ensure that the user is neither left idle, nor overwhelmed by an excess of data.

2.2 False Positives and Negatives

Ideally, early discard would reject all of the irrelevant data at the storage device without eliminating any of the desired data. This is impossible in practice for two reasons. First, the amount of computation available at the storage device may be insufficient to perform all of the necessary (potentially expensive) application-specific computations. Second, there is a fundamental trade-off [10] between false-positives (irrelevant data that is not rejected) and false-negatives (good data that is incorrectly discarded); in the information retrieval literature, this is known as the *precision/recall* tradeoff. Early discard algorithms can be tuned to favor one at the expense of the other, and different domain applications will make different trade-offs. For instance, an advertising agency searching a large collection of images may wish to quickly find a photo that matches a particular theme and may choose aggressive filtering; conversely, a homeland security analyst might wish to reduce the chance of accidentally losing a relevant object and would use more conservative filters (and accept the price of increased manual scanning). It is important to note that early discard does not, by itself, impact the accuracy of the search application: it simply makes applications that filter data more efficient.

2.3 Hardware Evolution

The idea of performing specialized computation close to the data is not a new concept. Database machines[17, 7] advocated the use of specialized processors for efficient data processing. Although these ideas had significant technical merits, they failed, at the time, because designing specialized processors that could keep pace with the sustained increase in general-purpose processor speed was commercially impractical. Implementing early discard on specialized computing hardware would invite a similar fate.

More recently, the idea of an *active disk* [1, 19, 22], where a storage device is coupled with a general-purpose processor, has become popular. The flexibility provided by active disks is well-suited to early discard; an active disk platform could run filtering algorithms for a variety of search domains, and can support applications that dynamically adapt the balance of computation between storage and host as the location of the search bottleneck changes [2]. Over time, due to hardware upgrades, the balance of processing power between the host computer and storage system will shift. In general, a system should expect a hetero-

geneous composition of computational capabilities among the storage devices as newer devices may have more powerful processors or larger memory. The more capable devices could execute more demanding early discard algorithms, and the partitioning of computation between the devices and the host computer should be managed automatically. Analogously, when the interconnect infrastructure or host computer is upgraded, one may expect computation to shift away from the storage devices. In practice, the best partitioning will depend on the characteristics of the processors, their load, the type and distribution of the data, and the query. For example, if the user were to search a collection of holiday pictures for snowboarding photos, one could expect these to be clustered together on a small fraction of devices, creating hotspots in the system. These observations motivate Diamond's two primary mechanisms for supporting this diversity: (1) enabling applications to specialize the early discard code for each storage device's capabilities; and (2) adaptive partitioning of computation between the storage devices and the host computer based on run-time measurements.

2.4 The Structure of Search

Early discard simplifies from past active disk research efforts because it exploits restrictions inherent to the search domain. First, search tasks only require read-only access to data. This simplifies the design by allowing us to avoid locking complexity and ignore some of the security issues. Second, search tasks typically permit stored objects to be examined in any order. This order independence offers several benefits: easy parallelization of early discard within and across storage devices; significant flexibility in scheduling data reads and simplified migration of computation between the active storage devices and host computer because the data has no inherent ordering constraints. Finally, most search tasks do not require maintaining state between objects. This stateless property supports efficient parallelization of early discard, and simplifies the run-time migration of computation between active storage device and host computer.

3 Rationale for Early Discard

This section describes a simple analytic model for early discard. Let N be the number of active storage devices in the storage system.[1] Let s_s and s_h be the speeds (data items processed per second) of each storage device and host , respectively. Let β be the fraction of processing that is performed on each storage device. Let p_d be the fraction of data that is discarded by the application; in the traditional architecture, discard can only occur at the host computer; in Diamond, the discard can occur either at an active storage device (early discard), or at the host computer. Let B_n be the bandwidth of the interconnect between storage and computer.

[1] This discussion assumes homogeneous storage devices; these results generalize to heterogeneous storage confi gurations, but the notation becomes more complex.

We assume that the bottlenecks in the exhaustive search pipeline can occur either at the storage device, the interconnect, or at the host computer. Since a pipeline progresses at the rate of its slowest stage, the time needed to process a data item is the maximum of the time needed for each stage to process the item. More formally:

$$\begin{aligned} T_s &= \frac{\beta}{s_s N}, \\ T_n &= \frac{(1-\beta p_d)}{B_n}, \\ T_h &= \frac{(1-\beta)}{s_h}, \\ T &= \max(T_s, T_n, T_h), \end{aligned}$$

where T_s, T_n, T_h is the time needed to process the item at the storage device, network and host computer, respectively; and T is the time taken by the object to pass through the system (assuming no additional latency).

To get some intuition about the benefits of early discard, let us consider some scenarios where all of the parameters are fixed, with the exception of β (the fraction of the computation performed at the active storage device). Clearly, β=0 denotes the traditional model, where the storage device is incapable of early discard and β=1 is the extreme early discard scenario where all of the filtering required for the search application is performed by the storage devices. By varying β, we can generate a family of curves (see Figure 2). The lowest point on the T curve is the most efficient configuration of the pipeline; β_o denotes the partitioning of computation that achieves this goal.

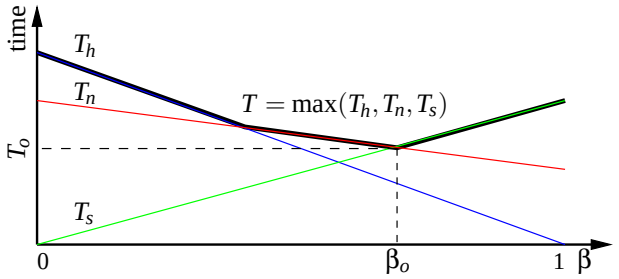


Figure 2. Processing time for exhaustive search with varying distribution of computation (β) between the host computer and active storage devices.

We can make several observations about this graph. Although the curves for the three stages will vary significantly based on parameter settings, they obey certain constraints: T_s goes through the origin and increases monotonically with β; T_h decreases monotonically with β, with T_h=0 at β=1; T_n decreases monotonically with β if the rejection rate for early discard is non-zero. These constraints imply that the optimal setting of β is always greater than zero — *i.e.*, early discard at the storage device is worthwhile even if the computational power of the storage device is significantly inferior to that

of the host computer. In practice, analytically determining β_o may not be feasible since the curves will change with different queries and data (both affect p_d). This motivates our desire to dynamically partition the search computation between host and storage devices, and to tune β during the progress of the exhaustive search.

4 Diamond Architecture

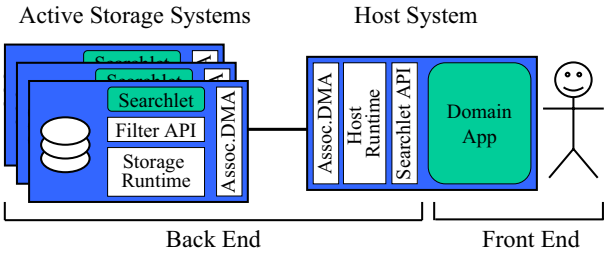


Figure 3. The Diamond storage architecture.

Figure 3 presents an overview of the Diamond storage architecture. Diamond provides a clear separation between the *front end*, which encapsulates domain-specific application code running on the host computer, and the *back end*, which consists of a domain-independent infrastructure that is common across a wide range of search applications.

In an interactive system, the user is a key component and one of the most likely bottlenecks. Diamond applications aim to reduce the load on the user by eliminating irrelevant data using domain-specific knowledge, ideally close to the storage device (early discard). Query formulation is domain-specific and is handled by the search application at the front end. Once a search has been formulated, the application translates the query into a set of machine executable tasks (termed a *searchlet*) for eliminating data, and passes these to the back end. The searchlet contains all of the domain-specific knowledge needed for early discard, and is a proxy of the application (and of the user) that can execute within the back end.

Searchlets are transmitted to the back end through the *searchlet API*, and distributed to the storage devices by the Diamond runtime system. At each storage device, the runtime system iterates through the objects on the disk (in a system-determined order) and evaluates the searchlet. The searchlet consists of a set of filters (see Section 6), each of which can independently decide to perform early discard. Any objects that pass through all of the filters in the searchlet are deemed to be interesting, and made available to the domain application through the searchlet API.

The domain application may perform further processing on the interesting objects to see if they satisfy the user's request. This additional processing can be more general than the processing performed at the searchlet level (which was constrained to the independent evaluation of a single object). For instance, this additional processing may include determining cross-object correlations and consulting auxiliary databases. Once the domain application determines that a particular object matches the user's criteria, the object is shown to the user. When processing a large data set, it is important to present the user with results as soon as they appear. Based on these partial results, the user can refine the query and restart the search. Query refinement leads to the generation of a new searchlet, which is once again executed by the back end. The system may be able to execute the refined search more efficiently by intelligently caching partial results from earlier searches.

4.1 Host System

The host system is where the domain application executes. The user interacts with this application to formulate searches and to view results.

In our implementation the storage devices may pass objects to the host without first evaluating the searchlet, due to resource limitations or other constraints. If the object is passed without evaluation, the host runtime will evaluate the searchlet before passing the object to the domain application. The host runtime and storage devices work in conjunction to balance computation between the host and the storage devices.

4.2 Searchlets

The searchlet is an application-specific proxy that the host and storage runtimes use to implement early discard. A searchlet consists of a set of *filters* and some configuration state (*i.e.*, parameters and thresholds for filters, and dependencies between filters). For example, a searchlet to retrieve portraits of people in dark business suits might contain two filters: a color histogram filter that finds dark spots and an image processing filter that locates human faces. A further filter that finds people with glasses may depend on having the face detector earlier in the pipeline.

For each object, the runtime will invoke each of the filters in an efficient order, considering both filter cost and selectivity (see Section 6.2). The return value from each filter indicates whether the object should be discarded, in which case the searchlet evaluation is terminated. If an object passes all of the filters in the searchlet, it is sent to the domain application.

The filter functions are sent as object code to Diamond. This choice of object code instead of alternatives such as domain-specific languages, was driven by several factors. First, many real-world applications (*e.g.*, drug discovery) may contain proprietary algorithms where requiring source code is not an option. Second, we want to encourage developers to leverage existing libraries and applications to simplify the development process. For instance, our image retrieval application (described in Section 5) relies heavily on existing image processing libraries such as OpenCV [8].

Executing application-provided object code on active storage devices does have serious security and safety implications. Diamond does not currently provide specific mechanisms to address these, but we believe that they can be solved using existing techniques such as software fault isolation [30] or virtual machines.

4.3 Key Interfaces

The Diamond architecture defines three APIs to isolate components: the *searchlet API*, the *filter API* and *Associative DMA*. These are briefly described below.

The searchlet API provides the interface that applications use to interact with Diamond. This API provides calls to query device capabilities, scope a search to a specific collection of data, load searchlets, and retrieve objects that match the current search.

The filter API defines the interface used by the filter functions to interact with the run-time environment on the storage device. This API allows functions to read and write object contents as well as setting attributes to annotate processed objects.

Associative DMA isolates the host and the storage implementations. We use this API to abstract the transport mechanism and flow control between host and storage runtime systems. Additionally, associative DMA provides a common interface that enables Diamond to employ diverse back end implementations without modifications to the host runtime. In addition, since large volumes of data are moved through the associative DMA API, it must be defined to enable efficient implementations that avoid data copies and redundant buffering.

4.4 Storage System

The storage system provides a generic infrastructure for filtering objects using searchlets. This infrastructure does not contain any domain-specific knowledge except for the searchlet.

We believe that search problems are well suited to active storage, but real active storage devices are not commercially available today. Furthermore, active storage systems will not become widely available without compelling applications to generate sufficient demand. Our approach is to provide a programming model that abstracts out the storage devices. This allows the development of applications that do not rely on active storage, but that will seamlessly exploit the active storage devices when they become available.

Diamond's current design assumes that the storage system can be treated as object storage [27]. This allows the host to be independent of the data layout on the storage device and will allow us to leverage the emerging object storage industry standards.

5 SnapFind

SnapFind is a prototype domain application built using the Diamond programming model. SnapFind was motivated by the observation that digital cameras allow users to generate thousands of photos yet few users have the patience to manually index them. Users typically wish to locate photos by semantic content (*e.g.*, "show me photos from our whale watching trip in Hawaii"); unfortunately, computer vision algorithms are currently incapable of understanding image content to this degree of sophistication. SnapFind's goal is to enable users to interactively search through large collections of unlabeled photographs by quickly specifying searchlets that roughly correspond to semantic content.

Research in image retrieval has attracted considerable attention in recent years [9, 12, 14, 20, 25, 28]. However, prior work in this area has largely focused on whole-image searches. In these systems, images are typically processed off-line and compactly represented as a multi-dimensional vector. In other systems, images are indexed offline into several semantic categories. These enable users to perform interactive queries in a computationally-efficient manner; however, they do not permit queries about local regions *within* an image since indexing every subregion within an image would be prohibitively expensive. Thus, whole-image searches are well-suited to queries corresponding to general image content (*e.g.*, "find me an image of a sunset") but poor at recognizing objects that only occupy a portion of the image (*e.g.*, "find me images of cell-phone users"). SnapFind exploits Diamond's ability to exhaustively process a data set using customized filters, enabling users to search for images that contain the desired content only in a small patch. The remainder of this section describes SnapFind, and then presents an informal validation of using early discard for this application.

5.1 Description

SnapFind currently employs several established image processing algorithms to filter image regions on the basis of color, texture or shape. Color is represented using image histograms [26, 13] and matched using an L1 similarity metric. We employ a coarse discretization of the color space for computational efficiency, but partially offset this by linearly interpolating pixel counts between adjacent bins. Texture is represented by the energy response of difference-of-gaussian filters [13] over the image patch. This simple filter is unable to capture any directional variation in texture but is computationally cheaper than more sophisticated alternatives such as Gabor jets or wavelets. Finally, we use the Viola-Jones [29] frontal face detector to identify regions that contain people.

Using SnapFind, users can create an initial query (*e.g.*, "find me images that contain a patch of blue color") and interactively refine the search based on partial search results (*e.g.*, "find me images that contain patches of blue color and texture that is like water"). Users can quickly construct their

own color and texture filters by selecting patches from previous search results (*e.g.*, the user may build a texture filter specialized for foliage by selecting regions in images containing plants).

5.2 Usage Experience

We designed some simple experiments to investigate whether early discard can help exhaustive search for a real-world problems. We chose the task of retrieving photos from an unlabeled collection. This is a realistic problem for owners of digital cameras and is also one that untrained users can perform manually (given sufficient patience). We explored two cases: (1) purely manual search, where all of the discard happens at the user stage; (2) using SnapFind. Both scenarios used the same Diamond interface (see Figure 4), where the user could examine six thumbnails per page, zoom in on a particular image (if desired) and mark selected images using the mouse.

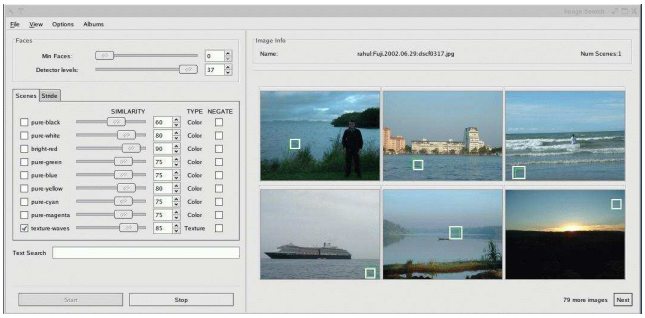


Figure 4. **SnapFind** , a proof-of-concept application designed using the Diamond programming model. Users can search a large image collection using customized filters. Unlike typical image retrieval systems that index the entire image, SnapFind enables users to find images that contain the desired features in small subregions. In this case, the user has filtered for regions containing water texture.

Our data set contained 18,286 photographs (approximately 10,000 personal pictures, 1,000 photos from a corporate website, 5,000 images collected from an ethnographic survey and 2,000 from the Corel image CD-ROMs). These were randomly distributed over twelve emulated active storage devices. Users were given three minutes to tackle each of the following two queries: (1) find images containing windsurfers or sailboats; (2) find pictures of people wearing dark business suits or tuxedos.

For the manual scenario, we recorded the number of images selected by the user (correct hits matching the query) along with the number of images that the user viewed in the alloted time. Users could page through the images at their own pace, and Table 1 shows that users scanned the images rapidly, viewing 600–1,000 images in three minutes. Even at this rate of 2–5 images per second, they were only able to process about 5% of the total data.

For the SnapFind scenario, the user built an early discard searchlet by selecting one or more image processing filters[2] Images that satisfied the filtering criteria (*i.e.*, those matching a particular color, visual texture or shape distribution in a subregion) were shown to the user. As in the manual scenario, the user then marked those images that matched the query. In addition to recording the number of hits and the number of images viewed by the user, we recorded the number of images that Diamond was able to process during the allotted time. The fraction of images discarded at the storage device was calculated, and these results are shown in Table 1.

	MANUAL		DIAMOND			
	good hits	user seen	good hits	user seen	system seen	early discard
Task 1						
user1	7	684	46	396	18,286	97.8%
user2	8	774	39	396	18,286	97.8%
user3	13	966	46	396	18,286	97.8%
Task 2(a)						
user1	29	600	42	588	18,286	96.8%
user2	18	612	31	426	18,286	97.7%
user3	24	630	32	474	18,286	97.4%
Task 2(b)						
user1	29	600	29	78	15,286	99.5%
user2	18	612	29	74	15,362	99.5%
user3	24	630	29	74	15,198	99.5%

Table 1. Results of an informal interactive search experiment using the SnapFind image search application. Users were given three minutes to locate photographs matching each query in a collection of 18,286 images.

In information retrieval, *recall* is the fraction of hits over the total number of desired objects in the data set while *precision* is the fraction of hits over the number of objects returned by the system.

For Task 1, the early discard searchlet was a single "water texture" filter trained on eight 32×32 patches containing water (listed as filter F5 in Table 3). For Task 2, users tried two searchlets: (a) a filter that simply looked for dark regions (listed as filter F3 in Table 3); (b) a conjunction of F3 and the face detector (filter F1). The former is a computationally-cheap searchlet favoring high recall, while the latter is a computationally-intensive searchlet that favors precision.

We can make several observations. On Task 1, SnapFind significantly increases the number of relevant images found by the user. Diamond is able to exhaustively search through all of the data, and early discard eliminates almost 98% of the objects at the storage devices. This shows how early discard can help users find an increased number of relevant objects.

On Task 2, the improvement, as measured by hits alone, is less dramatic, but early discard shows a different benefit. Task 2(b) highlights the advantages of an aggressive early

[2]Although SnapFind allows the user to interactively create specialized filters by presenting examples of matching image patches, we constrained the subjects in this experiment to a small set of pre-defined filters.

```
/* do search using filter_file and filt_spec */
do_search(filter_file, filt_spec)
{
  /* initialize the library */
  shandle = ls_init_search();

  /* set the searchlet for this search */
  ls_set_searchlet(shandle, DEV_ISA_IA32,
                   filter_file, filt_spec);

  /* start the search */
  ls_start_search(shandle);

  while (1) {
    err = ls_next_object(shandle, &cur_obj, 0);
    if (err == ENOENT) {
      /* the current search is done*/
      /* cleanup and return */
      ls_terminate_search(shandle);
      return;
    } else if (err == 0) {
      /* display the object, then release it */
      display_object(cur_object);
      ls_release_object(shandle, cur_obj);
      /* loop around and get the next object */
    } else {
      /* some unexpected condition */
      exit(1);
    }
  }
}
```

Figure 5. **Example Application Code** This pseudo code demonstrates the basic structure a Diamond application. The application initializes a search context, loads a searchlet, and iterates over the objects that match.

discard. Although the expensive filter fails to complete the exhaustive search in three minutes (it processes about 5/6 of the data), the results have a good precision; the user achieves approximately as many hits as in the manual scenario while viewing far fewer images (approximately 1/9 as many). For applications where the user only needs a few images, aggressive early discard is ideal because it significantly decreases the user's effort. A comparison between 2(a) and 2(b) illustrates the well-known trade-off between precision and recall. As the searchlet becomes more specialized, precision tends to improve at the cost of recall. For an interactive search, high precision may also come at the expense of increased wait time for the user — both because the filters are more expensive, and because a greater fraction of the stored data is discarded.

5.3 SnapFind Implementation

To build SnapFind, we implemented common image classification tasks as filters. The user formulates a query by selecting algorithms and representative image patches. Once a query is formulated, SnapFind generates a searchlet and begins a Diamond search. The pseudo code in figure 5 illustrates the basic steps for the search; the application initializes the library, sets a searchlet, and iterates over the matching objects. SnapFind dedicates a thread to interact with Diamond, and separate threads for the GUI and user interaction.

Figure 6 show portions of a filter specification for a query that looks for images of businessmen wearing glasses. There

```
# Find images that contain dark regions
FILTER      DarkPatch      # filter name
# Pass images that contain a patch that is at least
# 80% similar to the sample histogram.
THRESHOLD   80
FUNCTION    histo_find     # use color histogram fn
ARG         1.0            # first of list of args
ARG         64             # X patch size
ARG         64             # Y patch size
ARG         0.9211         # Histogram data
...                        # Histogram data
#
# This filter identifies regions in a image
# that contain faces.
FILTER      FaceDetect     # filter name
THRESHOLD   1              # pass if return >= 1
FUNCTION    face_detect    # use this function
ARG         1.0            # first of list of args
#
# Many more args, ...
#
# This filter determines whether a given face image
# contains glasses.
FILTER      Glasses            # filter_name
THRESHOLD   1                  # pass if return >=1
FUNCTION    glassses_recognize # use this function
ARG         64                 # one of the arguments
#
# Other arguments.
#
REQUIRES    FaceDetect      # Run face detect first
#
# ... other filters
#
```

Figure 6. **Example Filter Specification** This filter specification defines three filters. DarkPatch uses color histograms to find regions that match a sample histogram, FaceDetect looks for human faces, and Glasses looks for faces with glasses.

are three filters in this search; a patch of dark color (to indicates the presence of suits), a face detector, and a glasses detector.

The filter *DarkPatch* uses color histograms to search the image. In the example, the `histogram_find` function will return a value which is the closest similarity to the sample patch. In this example, the threshold is set to 80, raising this value would make the search more selective, and lowering the value would make it less selective. The arguments are interpreted by the filter function. Here, they specify the size of the image patch to test and the sample histogram.

The filter *FaceDetect* looks for human faces in the image. The threshold is 1, indicating at least one face must be found for the object to be considered interesting.

The last filter is *Glasses* which looks for glasses in pictures of human faces. Again the threshold is 1 indicating at least one instance of glasses must be found. The *Glasses* filter also uses the *REQUIRES* statement to indicate the *FaceDetect* filter must run before this filter.

Figure 7 gives the pseudo code for the *Glasses* filter. This code reads the *NUM_FACES* attribute (set by *FaceDetect*) to find the number of faces in the image. The filter then reads attributes that correspond to the bounding box of each face and tests these regions for glasses. It then returns the number of regions where glasses were found (and there are faces).

```
/* Processes bounding boxes containing faces to see
 * whether the image contains glasses.
 */
int
glasses_recognize(ohandle, void *conf_data, int len)
{
  char name_buffer[MAX_SIZE];
  num_found = 1;

  /*
   * read attribute to get the number of faces in
   * this image.
   */
  lf_read_attr(ohandle, "NUM_FACE", &bsize, &count);

  for (i=0; i < count; i++) {
    /* For each face, read the bounding box.  */
    sprintf(name_buffer, "%s%d\n", "FACE_BBOX", count);
    lf_read_attr(ohandle, name_buffer, &bbox, &size);

    /* see if image contains glasses */
    match = glasses_find(ohandle, conf_data, bbox);
    if (match) {
      /* keep track of the number of hits */
      num_found++;
    }
  }
  /* return the number of hits, different searches
   * could threshold against this number if they
   * wanted more than one person with glasses.
   */
  return(num_found);
}
```

Figure 7. **Example Filter Function** This pseudo code demonstrates the basic flow of a filter function. This filter function reads attributes that were set by a previous filter. The glasses detector uses these attributes to test regions only where there are known faces.

6 Prototype Implementation

Our Diamond prototype is currently implemented as a user process running on Red Hat Linux release 9.0. The searchlet API and the host runtime are implemented as a library that is linked against the domain application. The host runtime and network communication are threads within this library. We emulate active storage devices using off-the-shelf server hardware with locally-attached disks. The active storage system is implemented as a daemon. When a new search is started, new threads are created for the storage runtime and to handle network and disk I/O. Diamond's object store is currently implemented as a library that stores objects as files in the native file system. Associative DMA is currently under definition. Currently, Diamond uses a wrapper library built on TCP/IP; we use customized marshalling functions to minimize data copies.

The remainder of this section focuses on two important aspects of our implementation: (1) techniques employed for run-time partitioning of computational resources; (2) algorithms for determining an efficient filter ordering.

6.1 Dynamic Partitioning of Computation

As discussed in Section 2, bottlenecks in exhaustive search pipelines are not static. Diamond achieves significant performance benefits by dynamically balancing the computational task between the active storage devices and the host processor. This is a key part of the *Associative DMA* portion of the Diamond API for controlling data flow between the storage devices and the hosts, and our current prototype compares an initial set of implementation choices.

The Diamond storage runtime decides whether to evaluate a searchlet locally or at the host computer. This decision can be different for each object, allowing the system to have fine-grained control over the partitioning. Thus, even for searchlets that consist of a single monolithic filter Diamond can partition the computation on a per-object basis to achieve the ratio of processing between the storage devices and the host that provides the best performance. The ability to make these fine-grained decisions is made possible by Diamond's assumption that objects can be processed in any order, and that filters are stateless. Without these assumptions, the system would be limited to either processing everything at the storage devices or everything at the host. These extreme settings would prevent the system from fully utilizing the available processing resources.

The current implementation supports two methods for partitioning computation between the host and the storage devices. The effectiveness of these methods in practice is evaluated in Section 7.3.

CPU Splitting In this method, the host periodically estimates its available compute resources (processor cycles), determines how to allocate them among the active storage devices and sends a message to each device. When the storage device receives this message, it estimates its own computational resources and computes the percentage of objects that should be processed locally. For example, if a storage device estimates that it has 100 MIPS and receives a slice of 50 MIPS from the host, then it should choose to process 2/3 of the objects locally and send the remaining (unprocessed) objects to the host. CPU splitting has a straightforward implementation: whenever the storage runtime reads an object, it randomly selects (with the appropriate probability) whether the object should be processed locally.

Queue Back Pressure (Queue BP) This method is based upon the observation that the lengths of queues between components in the search pipeline (see Figure 1) provide good information about overloaded resources. The algorithm, to take advantage of this information, is implemented as follows.

When objects are sent to the host, they are placed into a work queue that is serviced by the host runtime. If the queue length exceeds a particular threshold, the host refuses to accept new objects. Whenever the storage runtime has an object to process, it examines the length of its transmit queue. If the queue length is less than a threshold, the object is sent to the host without processing. If the queue length is above the threshold, the storage runtime evaluates the searchlet and processes the object. Using this algorithm, the storage device adapts the computation per object based on the current

availability of downstream components. When the host processor or network is a bottleneck, the storage device will perform additional processing on the data; this eases the pressure on downstream components until data resumes its flow. Conversely, if the downstream queues begin to empty, the storage runtime will aggressively push data into the pipeline to prevent the host from becoming idle.

6.2 Filter ordering

As discussed in Section 4, a Diamond searchlet consists of a set of filters, each of which can choose to discard a given object. We assume that the set of objects that pass through a particular searchlet is completely determined by the set of filters in the searchlet (and their parameters), and is invariant to the order in which these filters are evaluated.[3] However, the filter order can dramatically impact the *efficiency* with which Diamond can process a large amount of data.

Diamond tries to reorder the filters within a searchlet to run the most promising ones early. Note that the best filter ordering depends on the set of filters, the user's query and the particular data set. For example, consider a user who is searching a large image collection for photos of people in dark suits. The application may determine that a suitable searchlet for this tasks includes two filters (see Table 3): a face detector that matches images containing human faces (filter F1); and a color filter that matches dark regions in the image (filter F3). From the table, it is clear that F1 is more selective than F3, but also much more computationally expensive. Running F1 first would work well if the data set contained a large number of night-time photos (which would successfully pass F3). On the other hand, if the collection contained a large number of baby pictures, running F1 early would be extremely inefficient.

More formally, the effectiveness of a filter depends upon its selectivity (discard rate) and its resource requirements. The total cost of evaluating filters over an object can be expressed analytically as follows. Given a filter, Fi, let us denote the cost of evaluating the filter as $c(Fi)$, and its pass rate as $\mathrm{P}(Fi)$. In general, the discard rates for the different filters may be correlated (*e.g.*, if an image contains a patch with water texture, then it is also more likely to pass through a blue color filter). We denote the *conditional pass rate* for a filter Fi that is processing objects that successfully passed filters Fa, Fb, Fc by $\mathrm{P}(Fi|Fa,Fb,Fc)$. The total time to process objects through a series of filters $F0 \ldots Fn$ is given by:

$$\begin{aligned} C &= c(F0) + \mathrm{P}(F0)c(F1) + \mathrm{P}(F1|F0)\mathrm{P}(F0)c(F2) + \\ &\quad \mathrm{P}(F2|F1,F0)\mathrm{P}(F1|F0)\mathrm{P}(F0)c(F3) + \cdots \end{aligned} \quad (1)$$

which takes into account all the conditional probabilities.

[3] This assumption is reasonable given that Diamond assumes filters to be stateless.

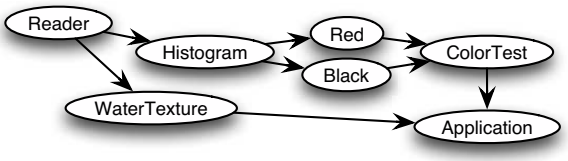


Figure 8. **Partial Order** - An example of a small partial ordering. "Reader" must be executed before "Histogram" and "WaterTexture". "Histogram" must be evaluated before "Red" and "Black".

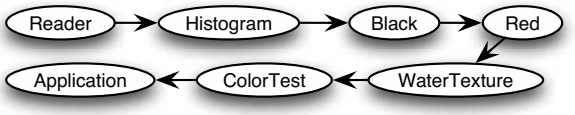


Figure 9. **Linear Extension** - One possible ordering of all the filters in Figure 8.

Partial Orderings

Allowing filters to use results generated by other filters enables searchlets to: (1) use generic components to compute well-known properties; and (2) reuse the results of other filters. For instance, all of the color filters in SnapFind (see Section 5) rely on a common data structure that is generated by an auxiliary filter. These dependencies can be expressed as *partial ordering* constraints. Figure 8 shows an example of a partial order. The forward arrows indicate an 'allows' relationship. For example, "Reader" is a prerequisite for "Histogram" and "WaterTexture", and "Red" and "Black" are prerequisites for "ColorTest". The filter ordering problem is then to find a *linear extension* of the partial order. Figure 9 shows one possible order. Note that finding a path through this directed acyclic graph is not sufficient; all filters still need to be evaluated.

Ordering Policies

The *filter ordering policy* determines the sequence in which filters are applied to each object. The ultimate goal is to minimizes the total cost of Equation 1. This cost depends not only on the inherent properties of the filters, but also on run-time parameters (user selectable "knobs") and the data itself. This means that knowing the optimal ordering (the **Offline Best** policy), *a priori*, is not possible We describe three techniques for online filter ordering:

- **Independent** If the filters are *independent* an optimal order can easily be generated by sorting [21]. If we assume that there is no correlation between $\mathrm{P}(Fi)$ and $\mathrm{P}(Fj)$, or $c(Fi)$ and $c(Fj)$, then the optimal order is given by sorting on $c(Fi)/\mathrm{P}(Fi)$. In practice, there will be correlation between filters and we consider several techniques to handle correlated filters below.
- **Hill climbing (HC)** picks a random starting point in the space of all legal linear extensions. It then tries to

incrementally improve the order by swapping adjacent filters. *Random restarts* reduce this technique's sensitivity to the starting point.

- **Best filter first (BFF)** leverages the structure of the cost function and accounts for dependencies between filters. It picks the filters F_i so as to minimize the corresponding cost term in Equation 1. Sorting can not be used to because of the dependent probabilities. Instead, we incrementally generate more and more restrictive partial orders, linearizing filters starting from the root of the original partial order ("Reader" in Figure 8). We can assign a minimum cost to these orders by computing the cost of the linear part. Our implementation uses a priority queue keyed on minimum cost to find the best partial orderings to extend. If the filters are truly independent, BFF reduces to the Independent policy.

7 Experimental Evaluation

This section presents empirical results from a variety of experiments using the Diamond system. All of these experiments employ the implementation described in Section 6 on the following hardware. The active storage platforms were emulated using rack-mounted 1.2 GHz Pentium III computers with 512 Mbytes of memory, and 73 GB SCSI disks, connected via a 1 Gbps Ethernet switch. The host system was a 3.06 GHz Pentium Xeon (8K L1 Dcache, 512K L2 cache) with 2 GB of memory, and 120 GB IDE disks. The host is connected via Ethernet to the rest of the storage platforms. We varied the link speed between 1 Gbps and 100 Mbps depending on the experiment. Some experiments required us to emulate slower active storage devices; this was done by running a real-time task that consumed a fixed percentage of the CPU

7.1 Description of Searchlets

We evaluate Diamond using the set of queries enumerated in Table 2. These include some real queries from SnapFind searches supplemented by synthetic queries designed to stress Diamond in different ways. The searchlets are briefly described in Table 2, and the filters used by these searchlets are listed in Table 3.

The Water and Business Suits queries match the tasks we used in Section 5. The Halloween query is similar to Business Suits with an additional filter. The three synthetic queries are used to test filter ordering and the two Dark Patch queries are used to illustrate bottlenecks for dynamic partitioning.

Table 3 provides a set of measurements summarizing the discard rate and the computational cost of running the various filters.

We determined these filter characteristics by evaluating each filter over every object in our image collection (described in Section 5). The overall discard rate is the fraction

Query		Searchlet Description	CPU Cost
Water - find image regions containing water waves	S1	Uses texture filter trained on water samples.	Low
Business Suits - find images of people in dark business suits	S2	Uses face detector and color histogram trained on dark patches of color.	High
Halloween - find images of a child in Halloween costume	S3	Uses face detector, color histogram trained on red patches of color, and color histogram trained on dark patches of color.	High
Synthetic	S4	Synthetic filters with inversely (non-linearly) related pass rate and cost.	Med
Synthetic	S5	Three filters with correlated pass rate and constant cost.	Low
Synthetic	S6	Three filters with independent pass rate (same as S5 overall) and constant cost.	Low
Dark Patch A - monolithic searchlet with high specificity	S7	Uses color histogram trained on black sample patch; has a high threshold so few images match.	Low
Dark Patch B - monolithic searchlet with low specificity	S8	Uses color histogram trained on black sample patch; has a low threshold so many images match.	Low

Table 2. **Test Queries** The queries and associated searchlets used to evaluate the Diamond prototype. With the exception of the synthetic queries, the searchlets were generated using SnapFind.

Filter -	Searchlet	Discard rate	CPU cost	(Range)
F0 - Reader (required)	S1,2,3,7,8	0	5	(1-7)
F1 - Face Detect	S2,3	99%	530	(40-1020)
F2 - Histogram	S2,3,7,8	0	20	(1-24)
F3 - Black (requires F2)	S2	83%	2	(0.5-3)
F3a - Black (requires F2)	S7	99%	2	
F3b - Black (requires F2)	S8	78%	2	
F4 - Bright Red (req. F2)	S2	99%	2	(0.3-3)
F5 - Wave Texture	S1	95%	14	(2-17)
F6 - Synthetic		20%	2	
F7 - Synthetic		22%	4	
F8 - Synthetic		26%	8	
F9 - Synthetic	S4	29%	16	
F10 - Synthetic		31%	32	
F11 - Synthetic		33%	64	
F12 - Synthetic		36%	128	
F13 - Synthetic		36%	256	
F14		50%	1	
F15 - Synthetic	S5	40%	8	
F16		30%	8	
F17		50%	1	
F18 - Synthetic	S6	40%	8	
F19		30%	8	

Table 3. **Filters** - The discard rate is over a collection of 18,286 images. Cost is measured in ms of computation time. Reader and Histogram are helper filters required by other filters. Filters in S5 are correlated, whereas those in S6 are not.

of objects dropped divided by the total number of images, and the cost is the average number of CPU milliseconds used to process that image with the filter shown. Filters F0-F5 are taken from SnapFind. The other filters were synthetically generated to have specific cost and discard characteristics.

The searchlets S5 and S6 were specifically designed to examine the effect of filter correlation; consequently F14,

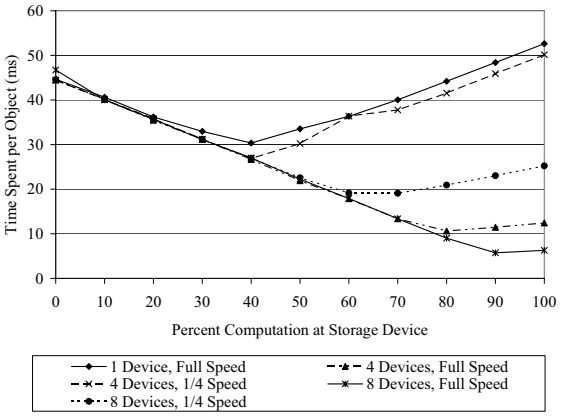


Figure 10. **Compute Limited** This experiments how the average time spent per object varies as we change the percentage of the objects evaluated at the storage system. The average time is computed as the total time to complete the search divided by the total objects searched.

F15 and F16 are correlated, whereas F17, F18 and F19 are not. For example, $P(F14, F15, F16) = 0.10$, while $P(F17, F18, F19) = P(F17)P(F18)P(F19) = 0.21$.

7.2 Comparing Disk and Host Processing Power

Our first measurements examine how variations in system characteristics (number of storage devices, interconnect bandwidth, relative processor performance, queries) affect the average time needed to process each object. We measure these variations by changing the relative processing power between the host and storage devices.

We also examine how varying the partition of work between the host and the storage device affects the search time. For each configuration, we measure the completion time for a different static partitioning between the host and storage devices. A particular partition is identified by percentage of objects that are evaluated at the storage devices. All remaining objects are passed to the host for processing.

In these experiments, each storage device has 5,000 objects (1.6 GB). As the number of storage devices increases, so does the total number of objects involved in a search. For each configuration we report the mean time needed for Diamond to process each object (averaged over three runs).

The first set of experiments look at how varying the hardware affects search time when the task is CPU bound. In these experiments we use searchlet Q8 to find pictures of a child in a Halloween costume. The results of these experiments are shown in figure 10.

The first observation is that, as the number of storage devices increases, more computation is moved to the storage devices. This matches our intuition that as the aggregate processing power of the storage devices increases, additional computation should be performed there.

When there is no processing at the storage devices this is equivalent to reading all the data from network storage. On

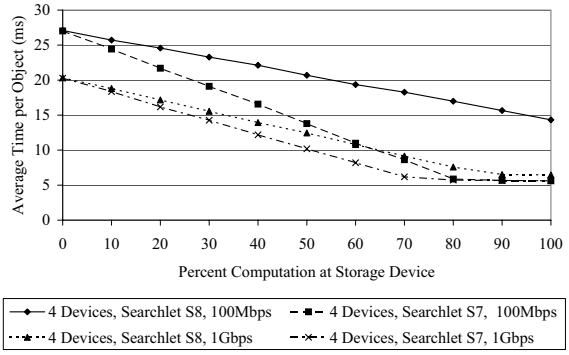


Figure 11. **Network Limited** This graph shows how the average time per object varies as we change the percentage of the objects evaluated at the storage system. The average time is computed as the total time to complete the search divided by the total objects searched.

the left-hand side of the lines we see linear decreases as processing is moved to the storage devices, reducing the load on the bottleneck. When most of the processing moves to the storage device, the bottleneck becomes the storage device, and we see increases in average processing time. This confirms what was predicted by the analytic model in Section 3. It is important to note that we gain a benefit from active storage even with a small number of storage devices.

Our next measurements examine what happens when the searchlet is network-bound instead of compute-bound. For these experiments we use searchlets Q7 and Q8. Both of these searchlets look for a small region of black in the images and are relatively cheap to compute. The difference is that the thresholds are varied so that Q7 will pass a small percentage of the objects (selective), and Q8 will pass a large percentage of the objects (non-selective).

In these graphs we see that in all cases the best solution is evaluating all the objects at the storage device. We also see that lines flatten out at some point. This point corresponds to the time it takes our current implementation to read the data off the storage device.

The upper two lines show Q7 and Q8 running on a 100 Mbps network. In this case we see that Q8 is always slower even when all computation is done at the storage device. This is because this searchlet passes a large percentage of the objects, so the data transfer to the host is the bottleneck. This clearly illustrates the benefit of selective filters that discard quickly.

7.3 Impact of Dynamic Partitioning

This section evaluates the effectiveness of the dynamic partitioning algorithms presented in section 6. These tests compare the best known static times to complete a search to the dynamically adjusted times using both the *CPU Splitting* and the *Queue BP* techniques.

For these tests we use both a CPU-bound task (searchlet Q2) and a network-bound task (searchlet Q7). We run each

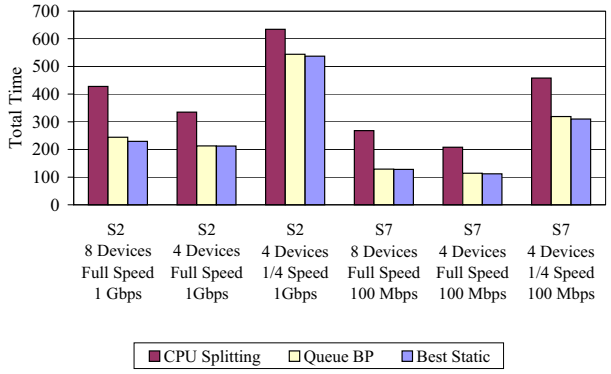


Figure 12. **Dynamic Partitioning** This graph compares the best performance using the manual partitioning and two different algorithms for automated partitioning. One algorithm uses the relative CPU power to split the load and the second uses queue back pressure.

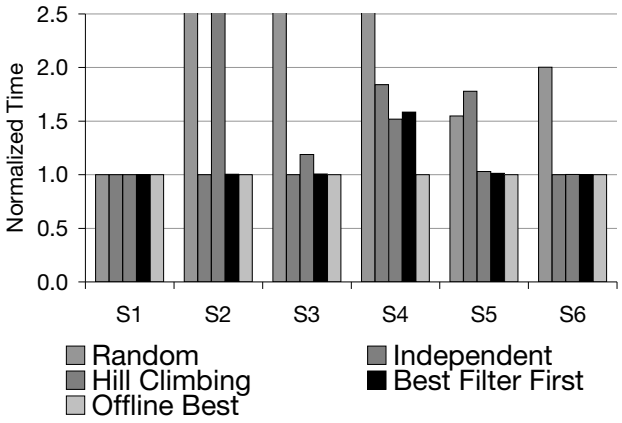


Figure 13. **Filter Ordering** Comparison of the time taken to evaluate various searchlets using different ordering policies. All of the times are normalized to the Offline Best policy.

task in a variety of configurations and compare the results as shown in Figure 12.

In all of these cases the Queue BP technique gives similar performance to the Best Manual technique. Surprisingly, the CPU Splitting does not do as well in most of the cases. We believe there are two reasons for this.

In the network-bound task (searchlet Q7), we know from the earlier sections that the best results are obtained by processing all objects at the storage devices. The CPU Splitting will always try to run some of the evaluation on the host, even when getting data to the host is the bottleneck. The Queue BP approach will see the bottleneck develop at the host and keep evaluating the searchlets locally.

The second observations is that relative CPU speeds are a poor estimate of time needed to evaluate the filters. Most of these searchlets involve striding over large data structures (the images), so the computation tends to be bound by memory access time, not computation. As a results scaling up the clock rate doesn't give a proportional decrease in time

It is possible that more sophisticated modeling would make the CPU Splitting more effective, but given that the simple Queue BP technique works so well, there is probably not much benefit to pursuing the idea.

7.4 Impact of Filter Ordering

This section compares the different policies described in Section 6.2, and illustrates the significance of filter ordering. We used searchlets S1-S6, which are composed of the filters detailed in Table 3. This experiment eliminates network and host effects by executing entirely on a single storage device and compares different local optimizations. Section 7.5 combines the effects of filter ordering and dynamic partitioning. Total time measurements are normalized to the *Offline Best* policy; this is the best possible static ordering (computed using an oracle), and provides a bound on the minimum time needed to process a particular searchlet. *Random* picks a random linear order at regular intervals. This is the simplest solution that avoids adversarial worst cases without extra state, and would be a good solution if filter ordering did not matter.

Figure 13 shows that the completion time can vary significantly with different filter ordering policies. (Different policies appear in the figure in the order mentioned below). The poor performance of *Random* demonstrates that filter ordering is significant. There is only one legal order for S1, so all methods pick the correct order. *Independent* finds the optimal ordering when filters are independent, as in S6, but can generate expensive orderings when they are not, as in S5. *Hill Climbing* has unpredictable performance, possibly because of its use of random starting points. *Best Filter First* is a dynamic techniques that works as well as *Independent* (it has a slightly longer convergence time) with independent filters, and has good performance with dependent filters. The dynamic techniques spend time exploring the search space, so they always pay a penalty over the *Offline Best* policy. This is more pronounced with more filters, as in S4.

7.5 Putting it Together

This section looks at running both filter ordering and the and dynamic partitioning together.

As a baseline we choose the optimal static partitioning for a given configuration and query. We then compare this to dynamically find the best partitioning and the best filter ordering. We also compare to using dynamic partitioning but a random filter ordering. The results of these experiments are shown in Figure 14.

The comparison shows that using dynamic partitioning with the Best Filter First ordering give comparable results to finding the Best Manual configuration. We see that using a Random ordering doesn't do as well. This is expected from the results in the previous section.

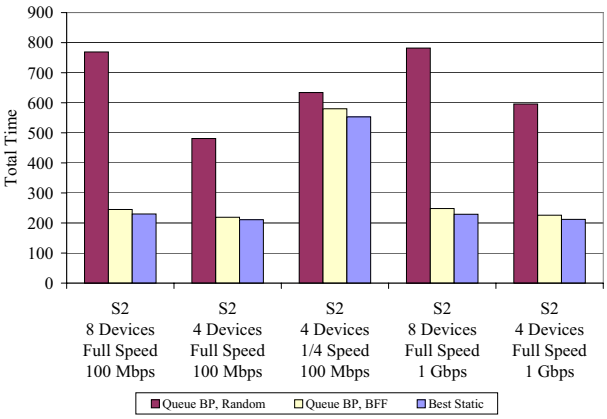


Figure 14. **Dynamic Optimizations** Comparison of the dynamic optimizations working together. The first case is best know static partitioning. The second case uses dynamic partitioning, but a random filter order. The third case uses the best first filter ordering as well as dynamic partitioning.

8 Related Work

Recent work on *interactive data analysis* [16] outlines a number of new technologies that will be required to make database systems as interactive as spreadsheets — requiring advances in databases, data mining and human-computer interaction. Diamond and early discard are complementary to these approaches, providing a basic systems primitive that furthers the promise of interactive queries.

In more traditional database research, advanced indexing techniques exist for a wide variety of specific data types including multimedia data [11]. Work on data cubes [15] takes advantage of the fact that many decision support queries are well-known to pre-process a database and then perform queries directly from the more compact representation. The developers of new indexing technology must constantly keep up with new data types, and with new user access and query patterns. A thorough survey of indexing and the outline of this tension appear in a recent dissertation [23], which also details theoretical and practical bounds on the (often high) cost of indexing.

Work on *approximate query processing*, most recently surveyed in [5] complements these efforts by observing that users can often be satisfied with approximate answers when they are simply using query results to iterate through a search problem, exactly as we motive in our interactive search tasks.

In addition, in high-dimensionality data (such as feature vectors extracted from images to support indexing) sequential scanning is often competitive with even the most advanced indexing methods because of the *curse of dimensionality* [31, 6, 10]. Indices do improve performance for low dimensionalities, or for queries on only a few attributes. However, in high dimensionality data and nearest neighbor queries, there is a lot of "room" in the address space and the data points are far from each other. The two major indexing methods, grid-based and tree-based, both suffer in high dimensionality data. Grid-based methods require exponentially many cells and tree-based methods group similar points together, resulting in groups with highly overlapping bounds. One way or another, a nearest neighbor query will have to visit a large percentage of the database, effectively reducing the problem to sequential scanning.

In systems research, our work builds on the insight of active disks [1, 19, 22] where the movement of search primitives to extended-function storage devices was analyzed in some detail, including for image processing applications. The work of Abacus [2], Coign [18], River [3] and Eddies [4] provide a more dynamic view in heterogeneous systems with multiple applications or components operating at the same time. Coign focuses on communication links between application components. Abacus automatically moves computation between hosts or storage devices in a cluster based on performance and system load. River handles adaptive dataflow control generically in the presence of failures and heterogeneous hardware resources. Eddies [4] adaptively reshapes dataflow graphs to maximize performance by monitoring the rates at which data is produced and consumed at nodes. The importance of filter ordering has also been the object of research in database query optimization [24]. The addition of early discard and filter ordering bring a new set of semantic optimizations to all of these systems, while retaining the basic model of observation and adaptation while queries are running.

Recent efforts to standardize object-based storage devices (OSD) [27] provide the basic primitives on which we build our semantic filter processing. In order to most efficiently process searchlets, active storage devices must contain whole objects, and must understand the low-level storage layout. We can also make use of the attributes that can be associated with objects to store intermediate filter state and to save filter results for possible re-use in future queries. Offloading space management to storage devices provides the basis for understanding data in the more sophisticated ways necessary for early discard filters to operate.

9 Conclusion

We have introduced the concept of *early discard* and analyzed it in the context of the Diamond interactive search system. Our goal is to build system support for *interactive data analysis* [16] where users are rapidly presented with (possibly incomplete) search results and are then kept "in the loop" in directing the search processing performed by the system. Early discard works by pushing filter processing to the edges of the system — executing semantic data filters directly at storage devices, and greatly reducing the flow of data into the central bottlenecks of a system. We have illustrated the utility of such a system in the context of image processing, specifically the extraction of images that meet a specific search criteria (*e.g.*, "people in business suits") which cannot be optimized by any indexing scheme today,

but only with user input (*i.e.*, "human in the loop search"). The filters at the storage devices are designed to prefer false positives over false negatives and let the user sort out the desired results. Multiple filters are run in series and may be re-ordered by the system depending on their (data dependent) selectivity and their (often data dependent) run-time performance.

We motivated the overall speedups possible through use of an analytic model and we presented an API for creating filters and searchlets. We provided an informal user study indicating that an early discard system can significantly aid a human user on certain search tasks. We described our prototype implementation, Diamond, and validated that it is robust across a variety of queries. We have shown that the system is able to dynamically adapt using queue pressure to a configuration as good as a manual placement, and better than one using estimated CPU capacity. We have also demonstrated that filter ordering within a searchlet can be determined dynamically and that the system can quickly arrive at a solution far superior to a naive ordering. The current Diamond system is an effective first step in realizing interactive search in unindexed data.

In future work, we plan to expand the data types and queries covered by the Diamond system and explore a more diverse set of filters for image processing. We expect to expand the sophistication of our user studies in order to strongly show that a system with early discard can satisfy users with both higher performance, and higher accuracy results than they could get by brute-force searching on their own. We also hope to encourage domain experts to use the Diamond programming framework to build real-world interactive search applications for other domains using the Diamond programming framework.

Acknowledgments

Thanks to D. Hoiem, B. Pillai for their valuable help with the Diamond system, and G. Bell and D. Westfall for providing some of the unlabeled data for the SnapFind user study.

References

[1] ACHARYA, A., UYSAL, M., AND SALTZ, J. Active disks: Programming model, algorithms and evaluation. In *Proc. of ASPLOS* (1998).

[2] AMIRI, K., PETROU, D., GANGER, G., AND GIBSON, G. Dynamic function placement for data-intensive cluster computing. In *Proc. of USENIX* (2000).

[3] ARPACI-DUSSEAU, R., ANDERSON, E., TREUHAFT, N., CULLER, D., HELLERSTEIN, J., PATTERSON, D., AND YELICK, K. Cluster I/O with River: Making the fast case common. In *Proc. of Input/Output for Parallel and Distributed Systems* (1999).

[4] AVNUR, R., AND HELLERSTEIN, J. Eddies: Continuously adaptive query processing. In *Proc. of SIGMOD* (2000).

[5] BABCOCK, B., CHAUDHURI, S., AND DAS, G. Dynamic sample selection for approximate query processing. In *Proc. of SIGMOD* (2003).

[6] BERCHTOLD, S., BOEHM, C., KEIM, D., AND KRIEGEL, H. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. of PODS* (May 1997).

[7] BORAL, H., AND DEWITT, D. Database machines: an idea whose time has passed? a critique of the future of database machines. In *Proc. of the International Workshop on Database Machines* (1983).

[8] BRADSKI, G. Programmer's tool chest: The OpenCV library. *Dr. Dobbs Journal* (November 2000).

[9] COX, I., MILLER, M., MINKA, T., PAPATHOMAS, T., AND YIANILOS, P. The Bayesian image retrieval system, PicHunter: theory, implementation and psychophysical experiments. *IEEE Transactions on Image Processing 9*, 1 (2000). Special Issue on Image and Video Processing for Digital Libraries.

[10] DUDA, R., HART, P., AND STORK, D. *Pattern Classifi cation*. Wiley, 2001.

[11] FALOUTSOS, C. *Searching Multimedia Databases by Content*. Kluwer Academic Inc., 1996.

[12] FLICKNER, M., SAWHNEY, H., NIBLACK, W., ASHLEY, J., HUANG, Q., DOM, B., GORKANI, M., HAFNER, J., LEE, D., PETKOVIC, D., STEELE, D., AND YANKER, P. Query by image and video content: the QBIC system. *IEEE Computer 28* (1995).

[13] FORSYTH, D., AND PONCE, J. *Computer vision: a modern approach*. Prentice Hall, 2002.

[14] GEMAN, D., AND MOQUET, R. Q & A models for interactive search, 2000. `http://www.math.umass.edu/~geman/Papers/qa.ps.gz`.

[15] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., AND VENKATRAO, M. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1* (1997).

[16] HELLERSTEIN, J., AVNUR, R., CHOU, A., HIDBER, C., RAMAN, V., ROTH, T., AND HAAS, P. Interactive data analysis: The Control project. *IEEE Computer* (August 1999).

[17] HSIAO, D. Database machines are coming, database machines are coming. *IEEE Computer 12*, 3 (1979).

[18] HUNT, G., AND SCOTT, M. The Coign automatic distributed partitioning system. In *Proc. of OSDI* (1999).

[19] KEETON, K., PATTERSON, D., AND HELLERSTEIN, J. A case for intelligent disks (IDISKs). *SIGMOD Record 27*, 3 (1998).

[20] MINKA, T., AND PICARD, R. Interactive learning using a society of models. *Pattern Recognition 30* (1997).

[21] PRUESSE, G., AND RUSKEY, F. Generating linear extensions fast. *SIAM Journal on Computing 23*, 2 (April 1994).

[22] RIEDEL, E., GIBSON, G., AND FALOUTSOS, C. Active storage for large-scale data mining and multimedia. In *Proc. of VLDB* (August 1998).

[23] SAMOLADAS, V. *On Indexing Large Databases for Advanced Data Models*. PhD thesis, University of Texas at Austin, August 2001.

[24] SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. Access path selection in a relational database management system. In *Proceedings SIGMOD* (1979).

[25] SMEULDERS, A., AND WORRING, M. Content-based image retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22*, 12 (2000).

[26] SWAIN, M., AND BALLARD, B. Color indexing. *International Journal of Computer Vision 7* (1991).

[27] Information technology – SCSI Object-Based Storage device commands (osd), September 2003. `http://www.t10.org/ftp/t10/drafts/osd/osd-r08.pdf`.

[28] TIEU, K., AND VIOLA, P. Boosting image retrieval. In *Proceedings of Computer Vision and Pattern Recognition* (2000).

[29] VIOLA, P., AND JONES, M. Rapid object detection using a boosted cascade of simple features. In *Proceedings of Computer Vision and Pattern Recognition* (2001).

[30] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Effi cient software-based fault isolation. In *Proc. of the 14th ACM Symposium on Operating System Principles* (December 1993).

[31] YAO, A., AND YAO, F. A general approach to D-Dimensional geometric queries. In *Proc. of STOC* (May 1985).